Zadanie domowe 6

Analiza żywotności kodu.

W tym zadaniu zajmiemy się zaimplementowaniem analizatora żywotności dla programów pisanych w MIPS-ie.

Załóżcie nowy katalog **lab6** i skopiujcie do niego całą zawartość katalogu **lab5**. Następnie ściągnijcie plik

`www.math.us.edu.pl/~pgladki/teaching/2013-2014/tk_lab6.zip`

i rozpakujcie go; w szczególności znajdziecie tam pliki graph.sig, graph.sml i liveness.sml. Zmodyfikujcie w odpowieni sposób plik sources.cm.

- Zaimplementujcie funkcję analyze w liveness.sml.
- Dodajcie linię kodu do Waszego pliku compile.sml która będzie wywoływać analizę żywotności dla każdej funkcji w Waszym Mips.program (w szczególności main, alloc, _printint).
- Zdebugujcie poprzez sprawdzenie wyjścia diagnostycznego drukowanego przez Liveness.interference_graph
- Prześlijcie mailem pliki README oraz liveness.sml. Termin zadania mija w **czwartek, 23 stycznia.** Proszę pamiętać o oznaczeniu maila tagiem `[aghtk]`.

Moduł Graph

Przeczytajcie dokładnie plik graph.sig. Zwróćcie uwagę, że różni się dość istotnie od pliku opisanego w Section 10.2 w podręczniku.

```
signature GRAPH = sig

structure S : ORD_SET

type node = S.Key.ord_key

type graph

val succ: graph ->node ->S.set

val pred: graph ->node ->S.set

val adj: graph ->node ->S.set

val newGraph: unit ->graph

val mk_edge: graph ->from: node, to: node ->unit

val rm_edge: graph ->from: node, to: node ->unit

val nodes: graph ->S.set

end
```

Grafy bazują na abstrakcji ORD_SET z biblioteki SML/NJ. Węzeł to po prostu dowolny typ porządkowy, którego relacja porządkująca dana jest przez S.compare. Dla grafów interferencji rejestrów, używamy node=Mips.reg (to znaczy S=Mips.RegSet).

Niech IG będzie wystąpieniem sygnatury GRAPH w której IGS=Mips.RegSet, to znaczy mamy dany graf rejestrów. W ML-u opisujemy go następująco:

`structure IG: GRAPH where S=Mips.RegSet`

i na tym etapie powinniście zauważyć, że dokładnie ta linia pojawia się w sygnaturze LIVENESS.

Aby skonstruować strukturę IG, stosujemy funktor OrdGraph z graph.sml w następujący sposób:

`structure IG = Graph(Mips.RegSet`

i możecie zauważyć, że zostało to już zrobione w liveness.sml.

Mając dany graf g możemy skonstruować zbiór wszystkich jego węzłów $IG.nodes(g)$. Możemy teraz uzyskać zbiór wszystkich następników węzła n (to znaczy węzłów x, dla których istnieje krawedź od n do x) wywołując $IG.succgn$. Podobnie można wywołać zbiór wszystkich poprzedników węzła n poprzez $IG.predgn$, oraz wszystkie węzły sąsiadujące (czyli następniki razem z poprzednikami) przez $IG.adjgn$, co z kolei jest równoważne wywołaniu $IG.S.union(IG.predgn, IG.succgn)$ (i, przy okazji, $IG.S.union$ to to samo, co Mips.RegSet.union).

Grafy mogą być modyfikowane destruktywnie za pomocą operatorów mk_edge oraz rm_edge. Chcąc stworzyć nowy, pusty graf, wywołujemy $IG.newGraph()$. Chcąc teraz dodać krawędź (x, y), wywołujemy $IG.mk\_edge\{from = x, to = y\}$; jeśli krawędź taka już istnieje, wywołanie to nie będzie miało żadnego efektu. Chcąc z kolei usunąć krawędź (x, y) wywołujemy $IG.rm\_edge\{from = x, to = y\}$ – i, podobnie, jeśli nie ma już tej krawędzi, wywołanie nie przyniesie żadnego efektu.

Pytania i odpowiedzi:

Pyt.: Dla grafów interferencji potrzebujemy przecież grafów nieskierowanych, a ten jest skierowany...

Odp.: Aby dodać krawędź, możecie wybrać dowolny kierunek. Dla zapytań wykorzystajcie funkcję adj.

Pyt.: Czy w grafie mogą pojawić się węzły, do których nie prowadzą żadne krawędzie?

Odp.: Tak.

Pyt.: Jak dodać węzeł, do którego nic nie prowadzi?

Odp.: Na przykład poprzez $IG.succgx$.

Pyt.: Ale to mało eleganckie...

Odp.: Cóż... `https://www.youtube.com/watch?v=mGBJg4GceHQ`

Moduł Liveness

Waszym zadaniem będzie zaimplementowanie analizy żywotności. W liveness.sml znajdziecie następującą sygnaturę:

```
signature LIVENESS = sig

structure IG: GRAPH where S=Mips.RegSet

val analyze: mention: Mips.reg ->unit,

interfere: Mips.reg ->Mips.reg ->unit ->

Mips.funcode ->unit

val interference_graph: Mips.funcode ->IG.graph
```

```
val printgraph: (string->unit) ->IG.graph ->unit

end
```

Funkcja *analyze* przyjmuje na wejściu listę podstawowych bloków i ma zgłosić wszystkie zmienne, jakie się pojawiają i wszelkie interferencje pomiędzy nimi. Robi to wywołując *mention* na każdym rejestrze i pseudorejestrze jaki jest w dowolnej chwili użyty przez dowolną instrukcję wewnątrz bloku oraz wywołując $interfere$ na każdej parze rejestrów które nie mogą być alokowane do tego samego rejestru.

Oczywiście można wielokrotnie wywoływać *mention* na tej samej zmiennej, czy też, odpowiednio, $interfere$ na tej samej parze zmiennych.

Przeanalizujcie funkcję $interference\_graph$ i zobaczcie, że działa na zasadzie wywoływania *analyze* z funkcją interferencji, która dodaje nową krawędź do grafu interferencji.

Algorytm

Zamiast używać Alogorithm 10.4 z podręcznika słowo w słowo, proponuję zastosować następującą wariację – przy czym lojalnie uprzedzam, że jest to tylko szkic, będziecie mieli sporo pracy i sporo decyzji do podjęcia dotyczących wykończenia szczegółów i optymalnego zaimplementowania Waszych pomysłów w ML-u.

Niech n przebiega po zbiorze etykiet bloków. Niech *live_at* będzie Symbol.table, która odwzorowuje etykiety na zbiory rejestrów (Mips.RegSet.set). Niech $block(n)$ będzie listą instrukcji, które składają się na blok etykietowany przez n. Niech $nextblock(n)$ będzie następną etykietą po n.

```
for each n

live_at[n] := {}

repeat

changed := false

for each n

new = compute_live_in(block( n ), live_at[nextblock( n )])

if new <>live_at[n] then changed := true

live_at[n] := new

until not changed
```

Porównując powyższe z Algorithm 10.4 zobaczycie, że $live\_at[n]$ spełnia tę samą fukncję, co $in[n]$. Algorytm $compute\_live\_in(il, live\_at\_end)$, gdzie il jest listą instrukcji, działa mniej więcej tak:

```
compute_live_in(i::rest, live_at_end) =

let live_out = compute_live_in(rest, live_at_end) in
```

```
if i is a straight-line instruction

then return (use(i) + (live_out - def(i)))

else if i is a conditional branch to label L

then return (use(i) + ((live_at[L] + live_out) - def(i)))

else if i is an unconditional branch to label L

then return (use(i) + (live_at[L] - def(i)))

compute_live_in(nil, live_at_end) =

return live_at_end
```

Use i def

Funkcja *Mips.instr_def_use* może okazać się bardzo pomocna, jako że jej zadaniem jest zliczanie use i def w instrukcjach.

Jal instruction def-use: Zauważmy, że *Mips.instr_def_use* nie raportuje w pełni użycia use i def w instrukcji *Jal*. Dzieje się tak, ponieważ te konkretne use i def zależą od tego, w jaki sposób klient *mips.sml* wybierze metodę implementacji konwencji procedura-wywołanie. Będziecie musieli potraktować tę instrukcję jak przypadek specjalny. Właściwie do zastosowanie use to rejestry actual-parameter ($a0), a właściwe def to rejestry caller-save, return-address i return-result ($v0).

System call def-use: Instrukcja $system - call$ MIPS-a ma różne use i def w zależności od wartości $v0. Sugerowałbym założyć, że instrukcja syscall jest zawsze bezpośrednio poprzedzana przez $load - immediate$ instrukcji ustalającej $v0. Następnie można zastosować taką sztuczkę:

```
compute_live_in(M.Li(r, i)::M.Syscall::rest, live_out) =>

if r = Mips.reg "$v0"

then (case M.syscall_def_use (M.immed2int i)

of SOME use,def =>(use + (live_out - def))

| NONE =>ErrorMsg.impossible "Unknown Syscall")

else ErrorMsg.impossible "Syscall not preceded by li $v0"
```

Zgłaszanie interferencji

Powyższy szkic algorytmu nie uwzględnia tego, że algorytm musi zgłaszać interferencje. Na stronie 222 w podręczniku (tuż przed Section 10.2) znajdziecie kryteria określające jakie interferencje należy zgłaszać.

Debuggowanie

Wydaje się, że najwygodniej jest "wypluć" zawartość *live_at* w następującym formacie:

```
################## LIVENESS: _gt

_gt: $a0 $s0 $s1 $s2 $s3 $s4 $s5 $s6 $s7 $ra

L4: $x27 $x28 $x29 $x30 $x31 $x32 $x33 $x34 $x35

L3: $x27 $x28 $x29 $x30 $x31 $x32 $x33 $x34 $x35 $x41

L6: $x27 $x28 $x29 $x30 $x31 $x32 $x33 $x34 $x35

L5: $x27 $x28 $x29 $x30 $x31 $x32 $x33 $x34 $x35 $x44

L2: $x27 $x28 $x29 $x30 $x31 $x32 $x33 $x34 $x35

L1: $x27 $x28 $x29 $x30 $x31 $x32 $x33 $x34 $x35 $x39

L8: $x27 $x28 $x29 $x30 $x31 $x32 $x33 $x34 $x35

L7: $x27 $x28 $x29 $x30 $x31 $x32 $x33 $x34 $x35 $x48

_gt.epilog: $v0 $s0 $s1 $s2 $s3 $s4 $s5 $s6 $s7 $ra
```